YILDIZ TEKNİK ÜNİVERSİTESİ

KİMYA ve METALURJİ FAKÜLTESİ

MATEMATİK MÜHENDİSLİĞİ BÖLÜMÜ

MTM2602 YAPAY ZEKAYA GİRİŞ DERSİ PROJE ÖDEVİ RAPORU

8-PUZZLE SOLVER

Ders Eğitmeni: Doç. Dr. Birol ARSLANYÜREK

Grup-3

CEMRE SENA EZGIN 22058051

SEMIH YUSUF KAYA 22058008

ANIL ESENER 22052072

EFE KARS 22052029

EMRE ŞAHIN ATAK 22058056

İstanbul, 2025

Problem Definition

The 8-Puzzle is a classic sliding-tile problem played on a 3×3 (could be 4x4 or 5x5 in this project) grid containing eight numbered tiles and one empty space. A legal move consists of sliding an adjacent tile (up, down, left, or right) into the space slot.

State Space
Each state is a permutation of the numbers 1 through 8 plus one blank (represented by 0), stored in a one-dimensional array of length nine.

Initial State
An arbitrary scrambled arrangement of the nine positions, provided by the user. For example: {{4,7,0},{8,1,6},{3,5,2}}.

Goal State
The ordered configuration: {{1,2,3},{4,5,6},{7,8,0}}.

Actions
Four possible moves:

UP: slide the tile below the blank upward

DOWN: slide the tile above the blank downward

LEFT: slide the tile to the right of the blank leftward

RIGHT: slide the tile to the left of the blank rightward

Transition Model
Applying an action produces a new state by swapping the blank with the specified neighboring tile. Each move has a step cost of 1.

Objective
Reach the goal state from the initial state using the minimum number of moves.

Environment Properties

- **Fully Observable**
  The agent has complete information about the current board configuration at every step.
- **Single-Agent**
  There is exactly one “agent” (the blank-tile mover) acting in the environment.
- **Deterministic**
  Each action (slide Up/Down/Left/Right) has a single, predictable outcome.
- **Sequential**
  Every move affects the future state; choices cannot be made independently.
- **Static**
  The board does not change unless the agent makes a move (no external dynamics).

- **Discrete**
  Both the state space (tile arrangements) and action space (four moves) are finite and discrete.

Data Types

Macros:

`PUZZLE_DIMENSION`
Defines the width/height of the square puzzle board. For the 8-Puzzle, this is set to 3 (i.e. a 3×3 grid).

`PUZZLE_SIZE`
The total number of cells on the board, computed as `PUZZLE_DIMENSION × PUZZLE_DIMENSION` (9 for a 3×3 puzzle).

`BLANK_TILE`
The integer value used to represent the empty space (the "blank") on the board. Here it is `0`.

`ACTIONS_NUMBER`
The number of possible moves (actions) in each state. With up/down/left/right, this is 4.

Enumeration Types:

`enum ACTIONS`
Lists the four legal moves of the blank space (i.e. sliding a neighboring tile):

`Move_Up` (slide the tile below into the blank)
`Move_Down` (slide the tile above into the blank)
`Move_Left` (slide the tile to the right into the blank)
`Move_Right` (slide the tile to the left into the blank)

`enum METHODS`
Enumerates all the search algorithms supported:

```
BreadthFirstSearch = 1,
UniformCostSearch = 2,
DepthFirstSearch = 3,
DepthLimitedSearch = 4,
IterativeDeepeningSearch = 5,
GreedySearch = 6,
AStarSearch = 7,
GeneralizedAStarSearch = 8
```

Structures:

`typedef struct State`
Represents a single puzzle configuration:

`int tiles[PUZZLE_SIZE]` (a flat (1D) array of length 9 storing tile numbers row-major)
`int blank_pos` (index (0–8) of the blank (0) in that array)
`float h_n` (heuristic value (e.g. Manhattan distance))
`unsigned int hash` (cached hash value for fast state comparisons)

`typedef struct Transition_Model`
Encapsulates the result of applying an action to a state:

`State new_state` (the state obtained after the move)
`float step_cost` (cost of the move (typically 1.0))

`typedef struct Node`
A node in the search tree:

`State state` (the puzzle configuration at this node)
`float path_cost` (g(n): cumulative cost from the root)
`enum ACTIONS action` (the move applied to the parent to get here)
`struct Node *parent` (pointer to the parent node (NULL for root))
`int Number_of_Child` (used by depth-first variants to track branching)

`typedef struct Queue`
A simple linked-list used as the frontier (open list):

`Node *node` (pointer to a search-node)
`struct Queue *next` (next element in the frontier)

Function Prototypes:

void Compute_State_Hash(State *state): Computes and stores a hash code in `state->hash` for quick lookup in the explored set.

State* Create_State(void): Prompts the user to enter a 3×3 board configuration and returns a newly allocated State.

void Print_State(const State *state): Prints the board in a human-readable 3×3 ASCII format.

void Print_Action(const enum ACTIONS action): Prints the name of the action (Move_Up, Move_Down, etc.).

int Result(const State *parent_state, const enum ACTIONS action, Transition_Model *trans_model): Applies `action` to `parent_state`, fills `trans_model` with the new state and cost and returns TRUE if the move is valid, FALSE otherwise.

float Compute_Heuristic_Function(const State *state,constState *goal_state): Computes and returns the Manhattan-distance heuristic h(n) (optionally with linear conflict)

int Goal_Test(const State *state, const State *goal_state): Returns TRUE if `state` matches `goal_state`, otherwise FALSE.

Node* First_GoalTest_Search_TREE(const enum METHODS method, Node *root, State *goal_state): Performs goal-test at node generation (used by BFS and Greedy).

Node* First_InsertFrontier_Search_TREE(const enum METHODS method, Node *root, State *goal_state, float alpha): Generalized search framework inserting into a priority frontier (used by UCS, A*, GenA*).

Node* DepthType_Search_TREE(const enum METHODS method, Node *root, State *goal_state, int Max_Level): Depth-first variants with optional depth limit (DFS, DLS, IDS).

Node* Child_Node(Node *parent, enum ACTIONS action): Allocates and returns a child node resulting from applying `action` to `parent`.

Queue* Start_Frontier(Node *root): Creates a new frontier (linked-list) initialized with `root`.

int Empty(const Queue *frontier): Returns TRUE if the frontier is empty.

Node* Pop(Queue **frontier): Removes and returns the front node from the frontier.

void Insert_FIFO(Node *child, Queue **frontier): Enqueues `child` at the end of the frontier (BFS).

void Insert_LIFO(Node *child, Queue **frontier): Pushes `child` onto the frontier (DFS).

void Insert_Priority_Queue_UniformSearch(Node *child, Queue **frontier): Inserts `child` into the frontier ordered by path-cost g(n).

void Insert_Priority_Queue_GreedySearch(Node *child, Queue **frontier): Inserts `child` into the frontier ordered by heuristic h(n).

void Insert_Priority_Queue_A_Star(Node *child, Queue **frontier): Inserts `child` into the frontier ordered by f(n)=g(n)+h(n).

void Insert_Priority_Queue_GENERALIZED_A_Star(Node *child, Queue**frontier, float alpha): Inserts `child` into the frontier ordered by f(n)=g(n)+α·h(n).

void Print_Frontier(const Queue *frontier): Prints the current contents of the frontier for debugging.

void Show_Solution_Path(Node *goal): Traces back from `goal` to the root, printing each state and action.

void Print_Node(const Node *node): Prints a single node (state, parent state, action, path_cost).

int Level_of_Node(Node *node): Returns the depth of `node` in the search tree.

void Clear_All_Branch(Node *node, int *allocated_count): Frees `node` and all its descendants, updating `allocated_count`.

void Clear_Single_Branch(Node *node, int *allocated_count): Frees only `node`, updating `allocated_count`.

void Warning_Memory_Allocation(void): Prints an error and exits if a `malloc` fails.

int Compare_States(const State *s1, const State *s2): Returns TRUE if two states have identical tile arrangements.

Node* Frontier_search(Queue *frontier, const State *state): Searches the frontier for a node matching `state`; returns pointer or NULL.

void Remove_Node_From_Frontier(Node *old_node, Queue **frontier): Removes 'old_node` from the frontier linked-list.

`void Generate_HashTable_Key(const State *state, unsigned char *key)`: Serializes a state into a string key for hashing.

`Hash_Table* New_Hash_Table(int size)`: Creates a new hash table of (prime) capacity ≥ `size`.

`void Resize_Hash_Table(Hash_Table *ht, int new_size)`: Resizes `ht` to a larger capacity, rehashing existing keys.

`void Delete_Hash_Table(Hash_Table *ht)`: Frees all memory associated with `ht`.

`void ht_insert(Hash_Table *ht, const State *state)`: Inserts `state` into the hash table (explored set).

`void ht_insert_key(Hash_Table *ht, const char *key)`: Inserts a precomputed key string into the hash table.

`int ht_search(Hash_Table *ht, const State *state)`: Returns TRUE if `state` is already in `ht`, otherwise FALSE.

`void Show_Hash_Table(Hash_Table *ht)`: Prints the current contents of the hash table (debug only).

Searching Algorithms

1) **Breadth-First Search:** An uninformed graph-search algorithm that systematically explores all states at increasing depths from the start state:

It uses a first-in, first-out (FIFO) queue called the frontier (open list).

Beginning at the root, it enqueues all valid child states, then dequeues the oldest node to expand next.

This “level-by-level” expansion guarantees that the first time it reaches the goal, it has found a shortest-path solution (in terms of move count) when all step-costs are equal.

BFS terminates as soon as the goal state is dequeued (or tested), making it complete and optimal for unweighted problems.

2) **Uniform-Cost Search:** An uninformed graph-search algorithm that always expands the least-cost node first.

It uses a priority queue (min-heap) as the frontier, ordered by path cost $g(n)g(n)g(n)$.

Starting from the root (with $g=0g=0g=0$), each time it dequeues the node with the smallest cumulative cost so far.

When it first dequeues the goal state, UCS is guaranteed to have found a least-cost (optimal) solution, even if step-costs vary.

UCS is complete (it will find a solution if one exists) and optimal for nonnegative step-costs, but can explore many nodes if costs are uniform (behaving like BFS in that case).

3) Depth-First Search: an uninformed graph-search algorithm that always expands the most recently generated node first:

It uses a last-in, first-out (LIFO) stack as the frontier.

Starting from the root, it pushes all children onto the stack and then repeatedly pops the top node to expand next.

DFS “dives” down one branch to its maximum depth before backtracking when it reaches a dead end (no unexpanded children).

Because it does not consider path cost or depth globally, DFS is not guaranteed to find a shortest-path solution (not optimal).

In infinite or cyclic state spaces, plain DFS may never terminate unless you enforce a depth limit or track explored states.

4) Depth-Limited Search: a variant of Depth-First Search that imposes a maximum depth bound:

It uses a last-in, first-out (LIFO) stack as the frontier, like DFS.

Parameter: a user-specified limit LLL.

Rule: when generating children, only push those whose depth ddd is strictly less than LLL.

If a node at depth LLL is reached, it is treated as a dead end (no further expansion).

Terminates when either the goal is found (at depth ≤L\le L≤L) or the frontier is exhausted.

Properties:

- Complete only if LLL is at least the depth of the shallowest solution.
- Not optimal in general (it follows one branch fully).
- Space is linear in LLL (unlike BFS's exponential growth).

5)Iterative Deepening Search:

Runs a series of depth-limited depth-first searches, each time increasing the depth limit by 1:

1. Do DFS with limit = 0.
2. If goal not found, do DFS with limit = 1.
3. Repeat with limit = 2, 3, … until the goal is reached.

- **Memory-efficient:** uses DFS's low memory.
- **Guarantees shortest path:** finds the shallowest solution first.
- **Simple to implement:** just wrap DFS in a loop that raises the limit.

6) Greedy Search:

Greedy Search is an uninformed search strategy that uses only the heuristic estimate to guide the search:

- It maintains a priority queue (the frontier) ordered by the heuristic value h(n)h(n)h(n) of each node.
- At each step, it selects and expands the node with the lowest h(n)h(n)h(n), i.e., the state that appears "closest" to the goal according to the heuristic.
- It ignores the path cost g(n)g(n)g(n) entirely, focusing solely on minimizing estimated remaining cost.
- Greedy Search is not guaranteed to find an optimal (shortest-path) solution, nor is it complete in infinite or cyclic spaces—however, it often finds a solution quickly if the heuristic is good.

7)A* Search:

an informed search algorithm that finds an optimal path by combining actual path cost and heuristic estimate:

It uses a priority queue ordered by the evaluation function

f(n)=g(n)+h(n) f(n) = g(n) + h(n)f(n)=g(n)+h(n) where

- g(n)g(n)g(n) is the cost from the start to node nnn,
- h(n)h(n)h(n) is the heuristic estimate from nnn to the goal.

At each step, A* expands the node with the lowest f(n)f(n)f(n), balancing between proven cost so far and estimated remaining cost.

If the heuristic h(n)h(n)h(n) is admissible (never overestimates), A* is complete and optimally finds the shortest-path solution.

8)Generalized A* Search:

Generalized A* extends the classic A* by adding a weight α\alphaα to the heuristic term:

- **Evaluation function:**

  f(n)=g(n)+α h(n) f(n) = g(n) + \alpha \, h(n)f(n)=g(n)+αh(n)

  where

  - g(n)g(n)g(n) is the cost so far,
  - h(n)h(n)h(n) is the heuristic estimate to goal,
  - α\alphaα is a user-defined weight.
- **Frontier management:**
  Uses a priority queue ordered by f(n)f(n)f(n). Nodes with lower f(n)f(n)f(n) are expanded first.
- **Behavior by α\alphaα:**
  - α=0\alpha = 0α=0 → Uniform-Cost Search (f(n)=g(n)f(n)=g(n)f(n)=g(n))
  - α=1\alpha = 1α=1 → Standard A* (f(n)=g(n)+h(n)f(n)=g(n)+h(n)f(n)=g(n)+h(n))
  - α>1\alpha > 1α>1 → Heuristic-biased search (more "greedy" toward goal)
- **Properties:**
  - If 0≤α≤10 \le \alpha \le 10≤α≤1 and h(n)h(n)h(n) is admissible, the algorithm is complete and optimal.
  - Larger α\alphaα can reduce expansions at the cost of potentially sacrificing optimality.

Deciding The Most Optimistic Heuristic Function

To determine which heuristic functions run fastest in the A* algorithm, we executed three different heuristics on two distinct start states and timed them with a stopwatch:

1. **Start State (Path Cost = 14):**
   {{1,2,3},{4,0,5},{6,7,8}}
   a) **Number of Misplaced Tiles:** 1 minute 45 seconds
   b) **Euclidean Distance:** 28 seconds
   c) **Manhattan Distance + Linear Conflict:** 7 seconds
2. **Start State (Path Cost = 22):**
   {{7,2,4},{5,0,6},{8,3,1}}
   a) **Number of Misplaced Tiles:** over 10 minutes
   b) **Euclidean Distance:** 2 minutes 17 seconds
   c) **Manhattan Distance + Linear Conflict:** 51 seconds

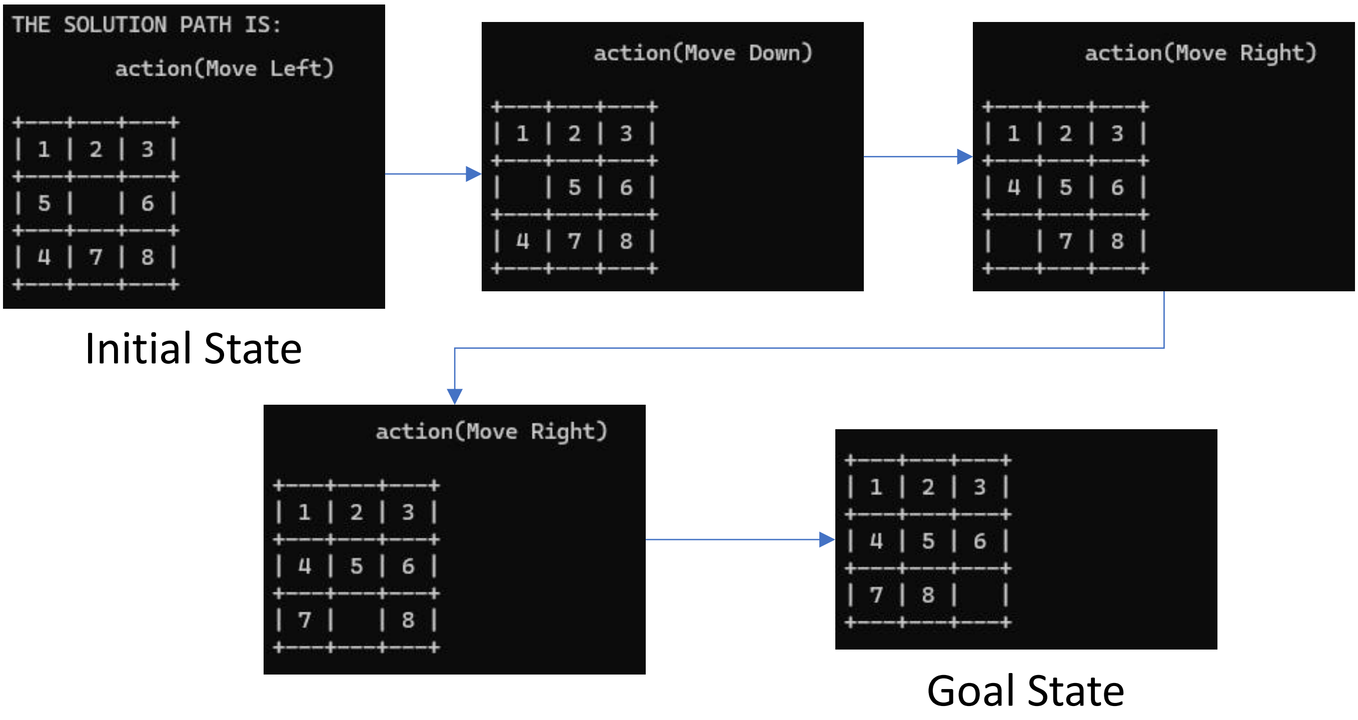
Based on these results, Manhattan Distance with Linear Conflict was chosen as the heuristic for our A* implementation.

Results

For 3x3 Board:

Assume that user entered an initial state {{1, 2, 3},{5, 0, 6},{4, 7, 8}} which requires 4 steps.

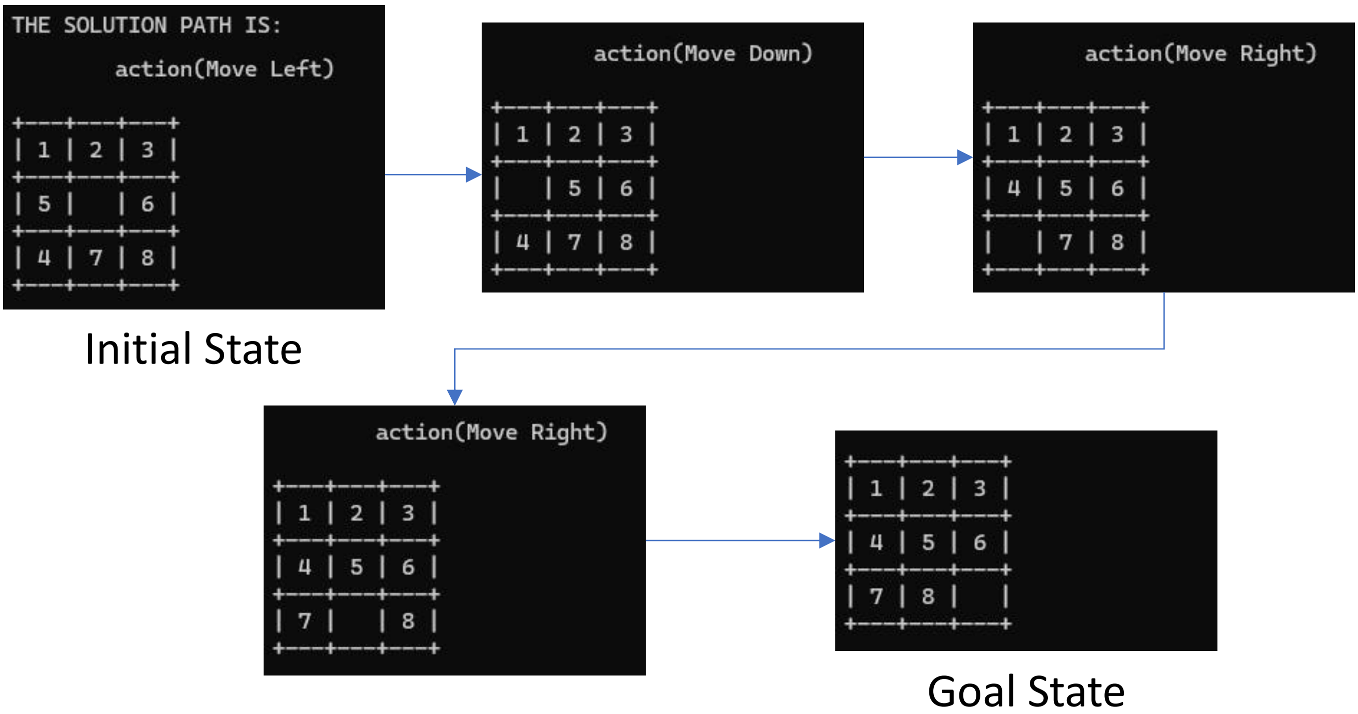
1) **Breadth-First Search:**

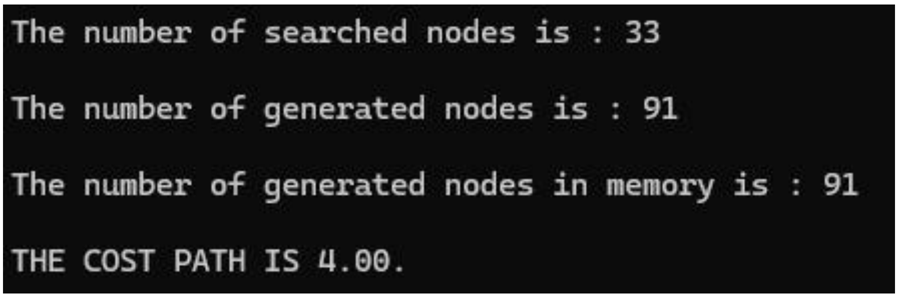



```
The number of searched nodes is : 33

The number of generated nodes is : 51

The number of generated nodes in memory is : 51

THE COST PATH IS 4.00.
```

It took 2,93 seconds to find the solution

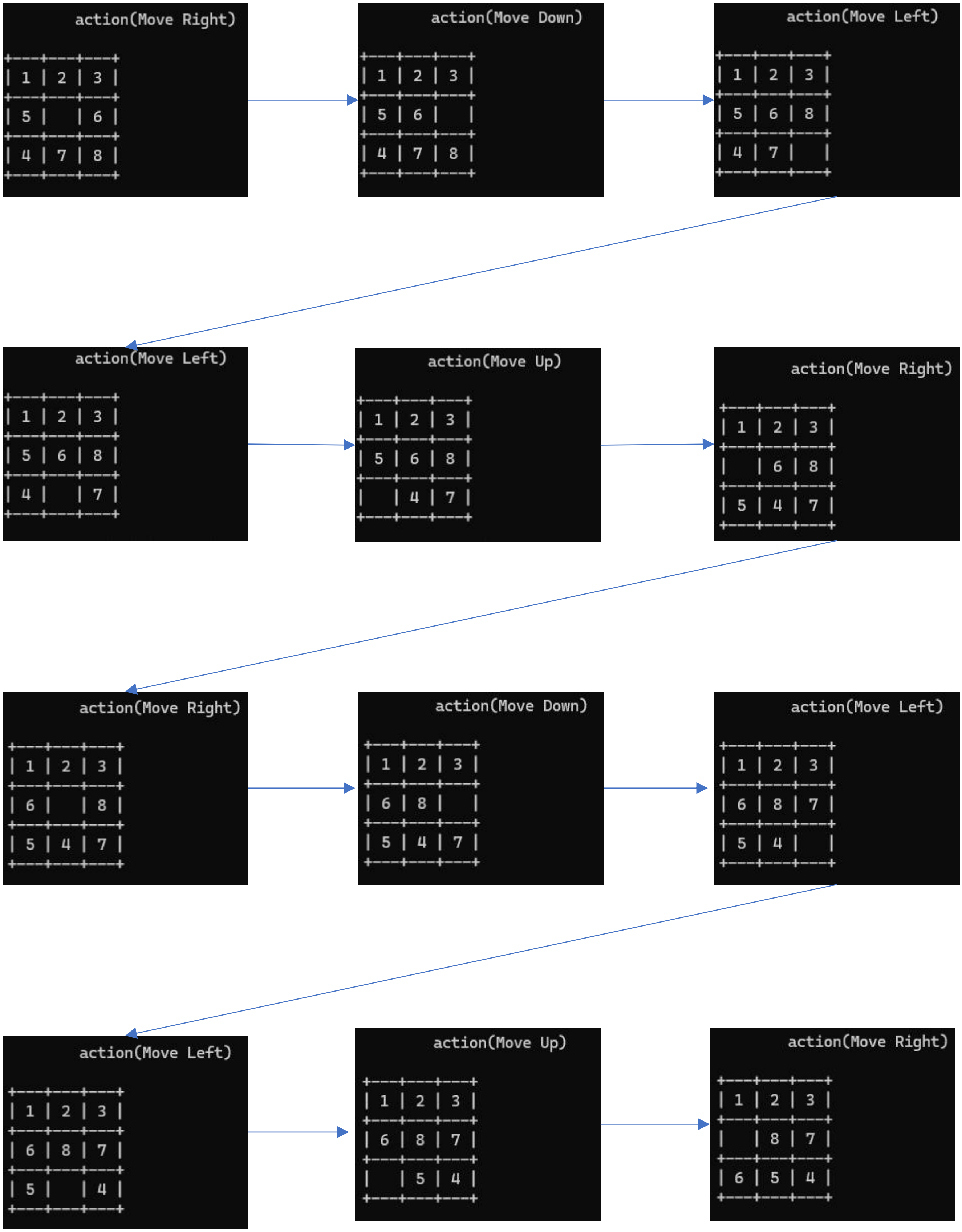
2) Uniform-Cost Search:

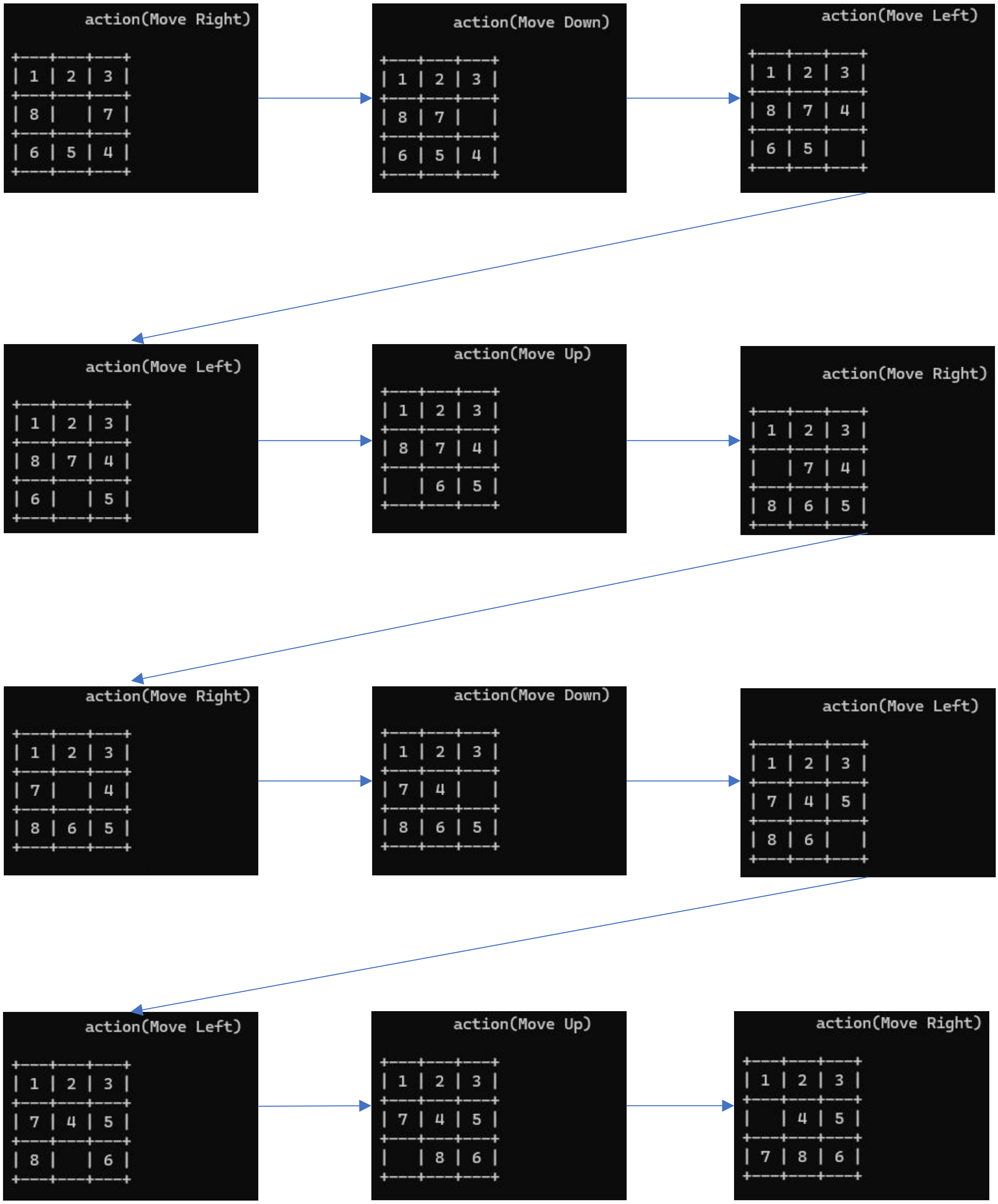


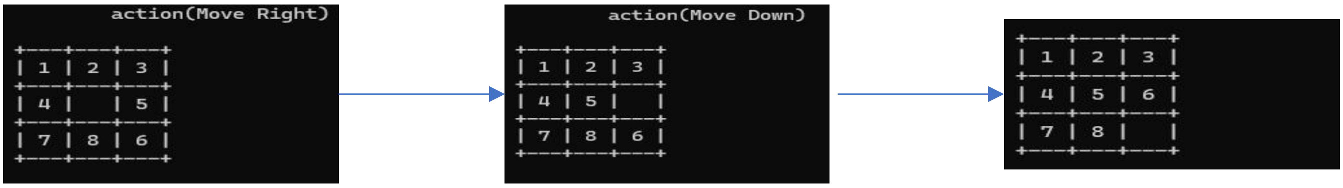

```
The number of searched nodes is : 33

The number of generated nodes is : 91

The number of generated nodes in memory is : 91

THE COST PATH IS 4.00.
```

It took 5.33 seconds to find the solution

3) Depth-First Search




action(Move Right)
1	2	3
8		7
6	5	4
action(Move Down)
1	2	3
8	7	
6	5	4
action(Move Left)
1	2	3
8	7	4
6	5	
action(Move Left)
1	2	3
8	7	4
6		5
action(Move Up)
1	2	3
8	7	4
	6	5
action(Move Right)
1	2	3
	7	4
8	6	5
action(Move Right)
1	2	3
7		4
8	6	5
action(Move Down)
1	2	3
7	4	
8	6	5
action(Move Left)
1	2	3
7	4	5
8	6	
action(Move Left)
1	2	3
7	4	5
8		6
action(Move Up)
1	2	3
7	4	5
	8	6
action(Move Right)
1	2	3
	4	5
7	8	6

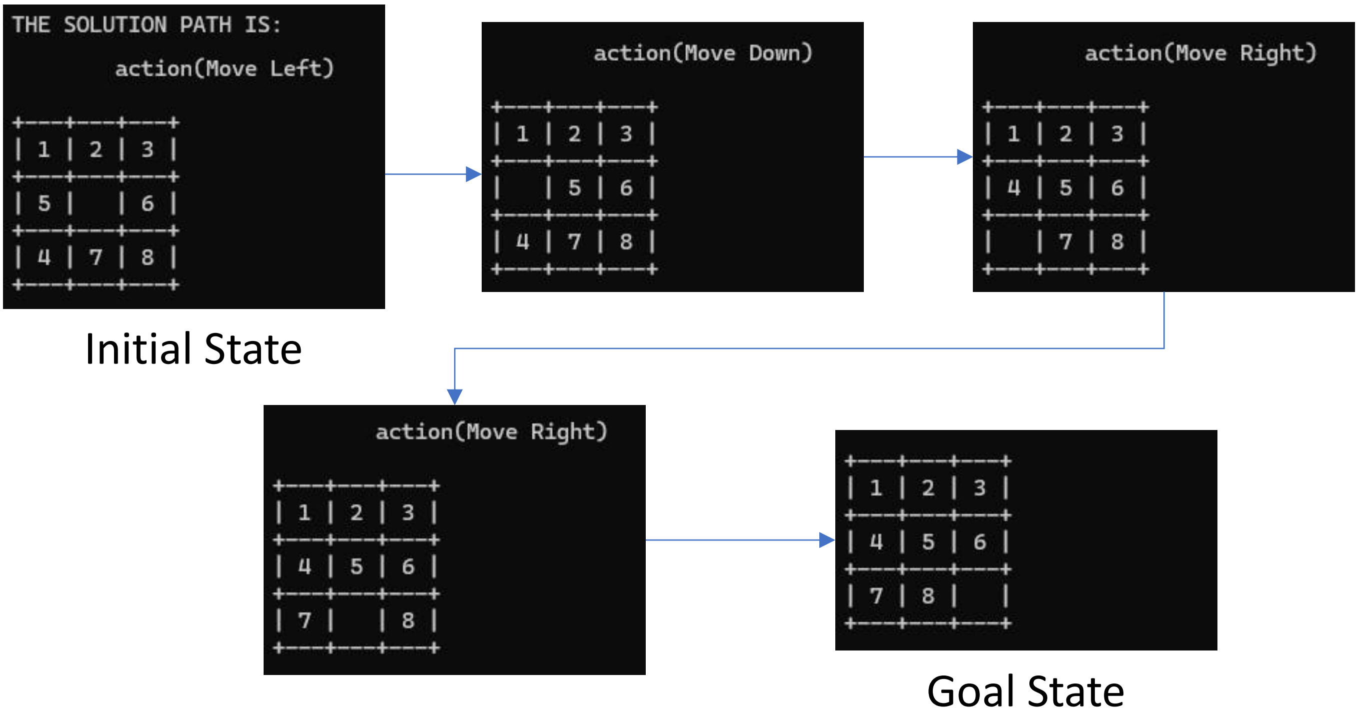



```
The number of searched nodes is : 51

The number of generated nodes is : 75

The number of generated nodes in memory is : 51

THE COST PATH IS 26.00.
```

It took 3.13 seconds to find the solution

4) Depth-Limited Search:

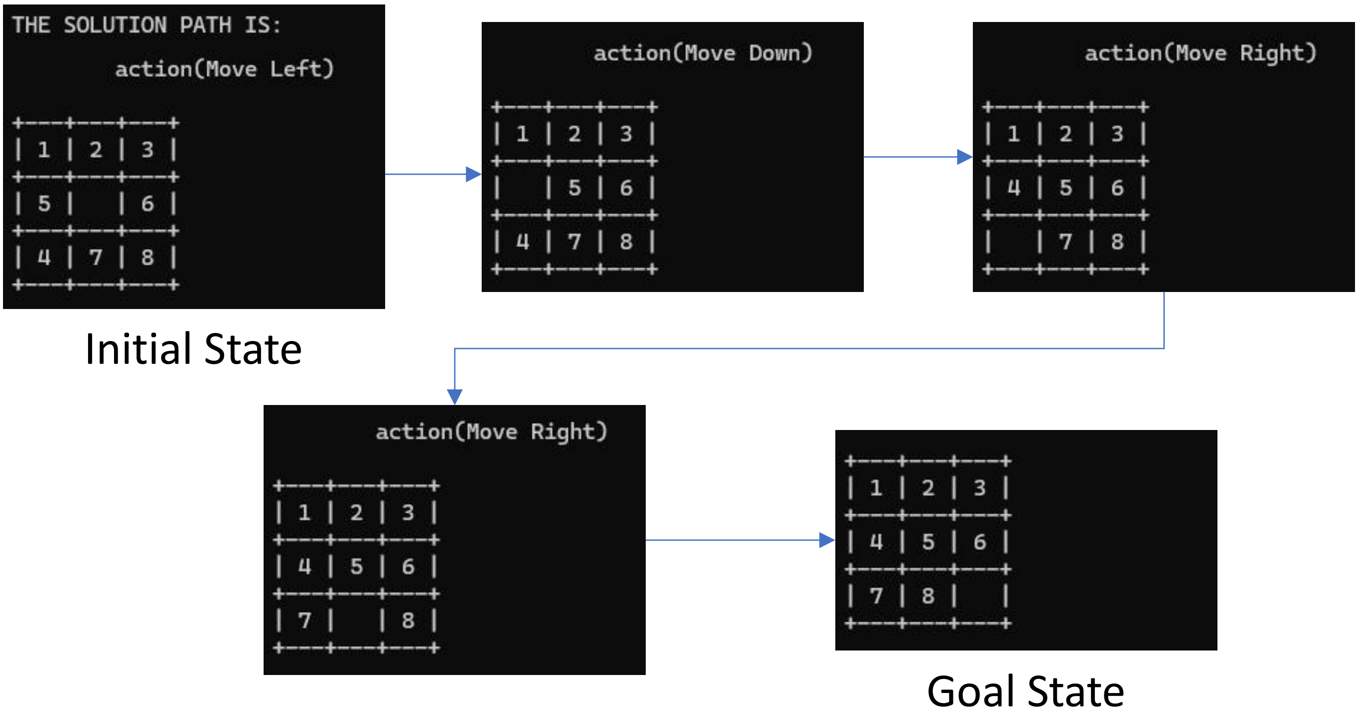
In this algorithm, selecting a depth of 3 or less will fail to find a solution because no solution exists at that depth. For the given example, a depth of 4 has been chosen.




```
The number of searched nodes is : 18

The number of generated nodes is : 26

The number of generated nodes in memory is : 9

THE COST PATH IS 4.00.
```

It took 2.11 seconds to find the solution.

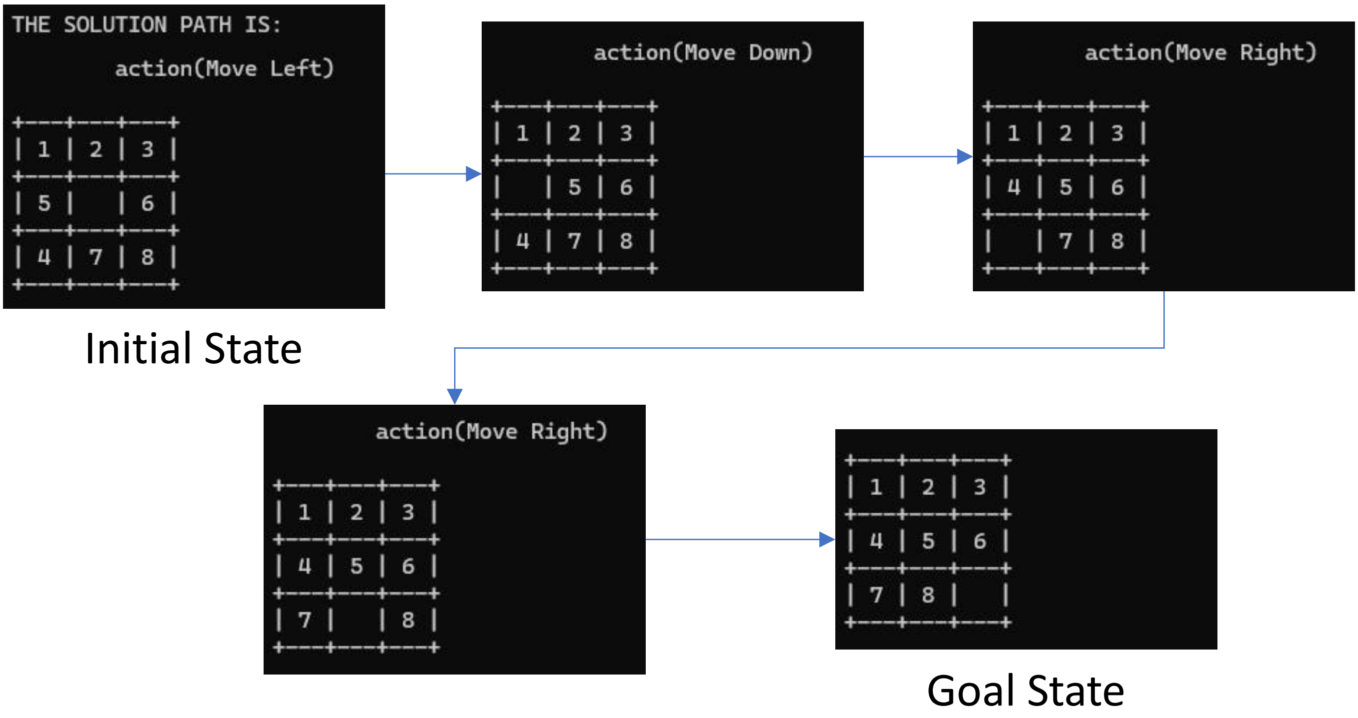
5) Iterative Deepening Search

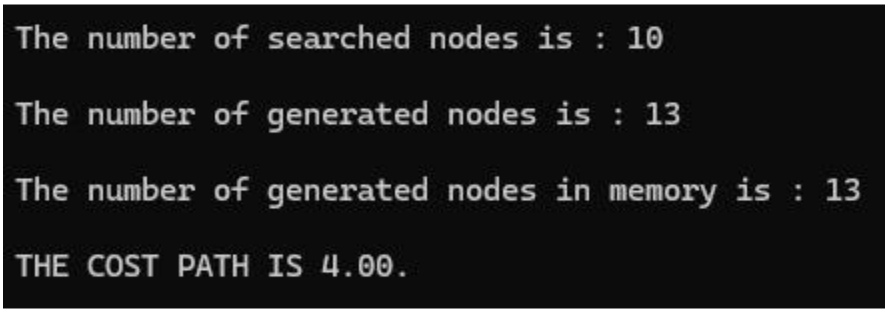



```
The number of searched nodes is : 58

The number of generated nodes is : 78

The number of generated nodes in memory is : 9
The goal is found in level 4.

THE COST PATH IS 4.00.
```

It took 6.08 seconds to find the solution.

6) Greedy Search




The number of searched nodes is : 10

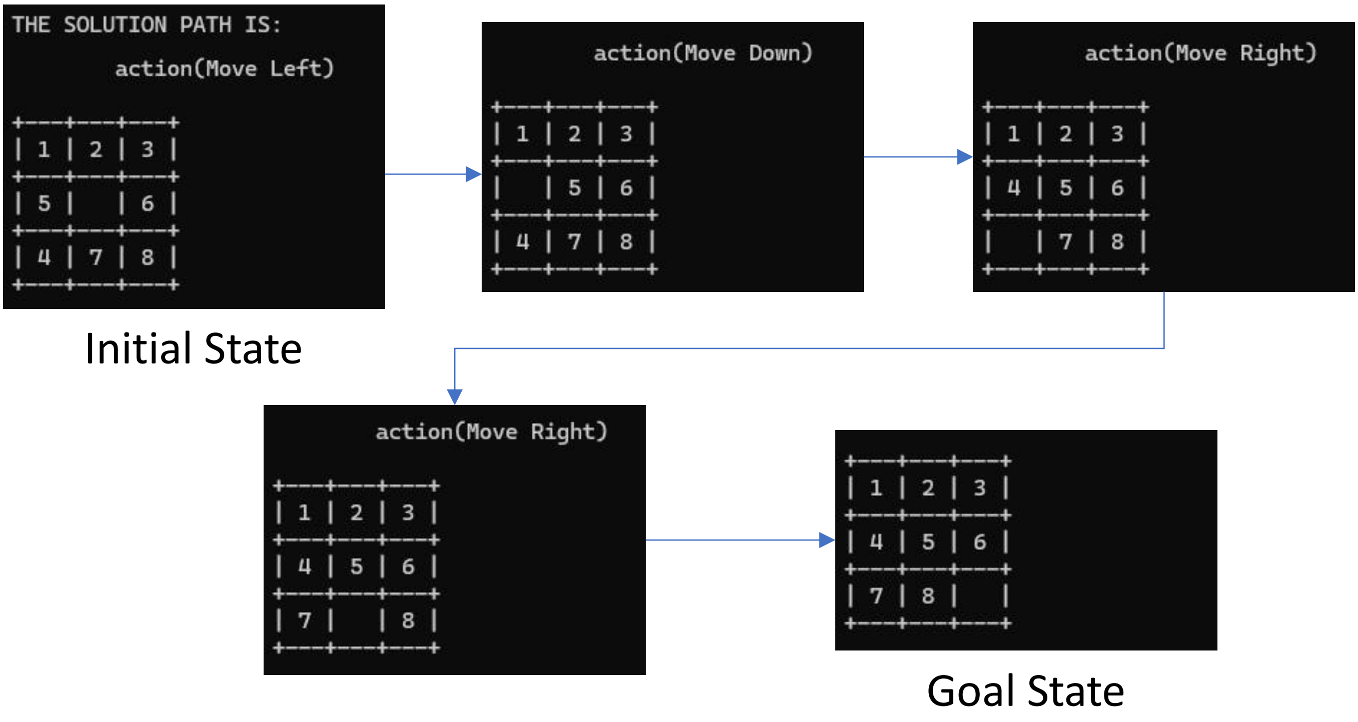
The number of generated nodes is : 13

The number of generated nodes in memory is : 13

THE COST PATH IS 4.00.

It took 1.19 seconds to find the solution.

7) A* Search

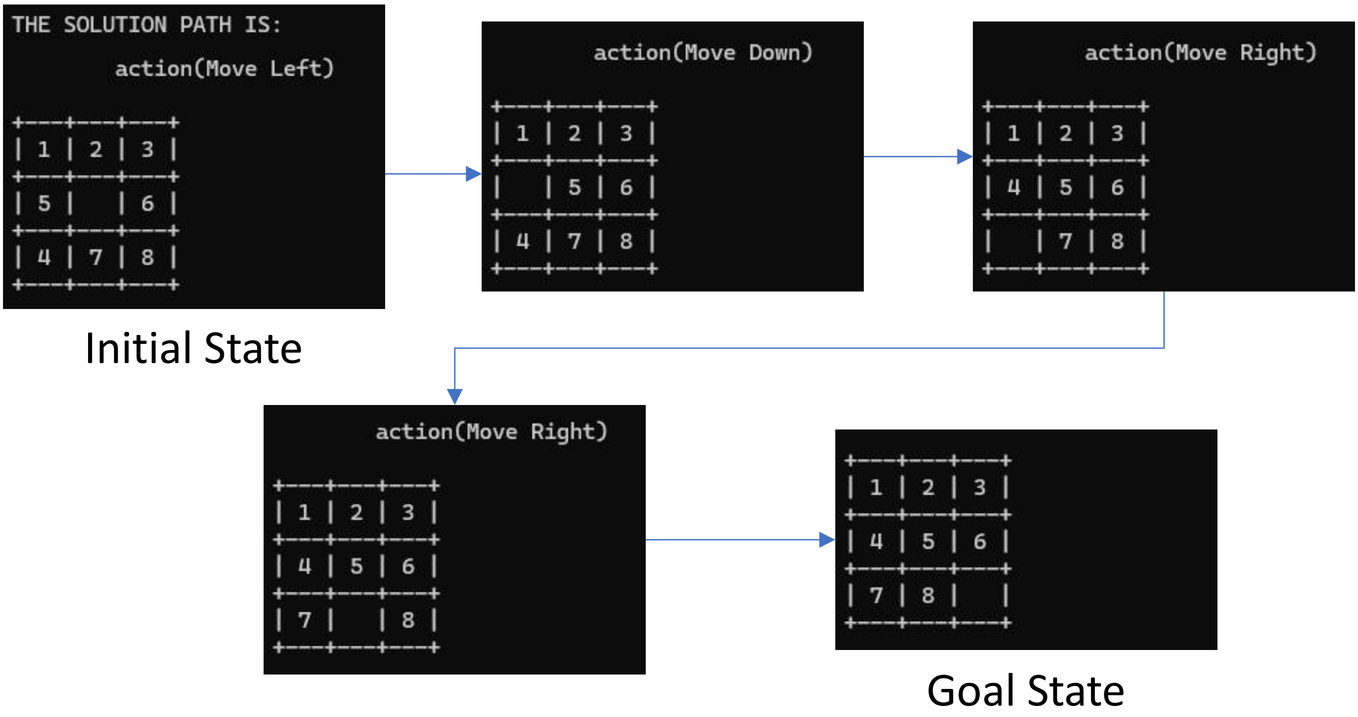



```
The number of searched nodes is : 5

The number of generated nodes is : 13

The number of generated nodes in memory is : 13

THE COST PATH IS 4.00.
```

It took 1.09 seconds to find the solution

8) Generalized A* Search:

In generalized A*, the **critical α value** is defined as the **smallest weight coefficient** that ensures the goal node’s $f\alpha(n) = g(n) + \alpha\, h(n)$ remains **lower than that of every other node** in the frontier—thereby guaranteeing that the goal node is expanded first. In this case, the critical α was found to be **0.61**.




```
The number of searched nodes is : 5

The number of generated nodes is : 13

The number of generated nodes in memory is : 13

THE COST PATH IS 4.00.
```

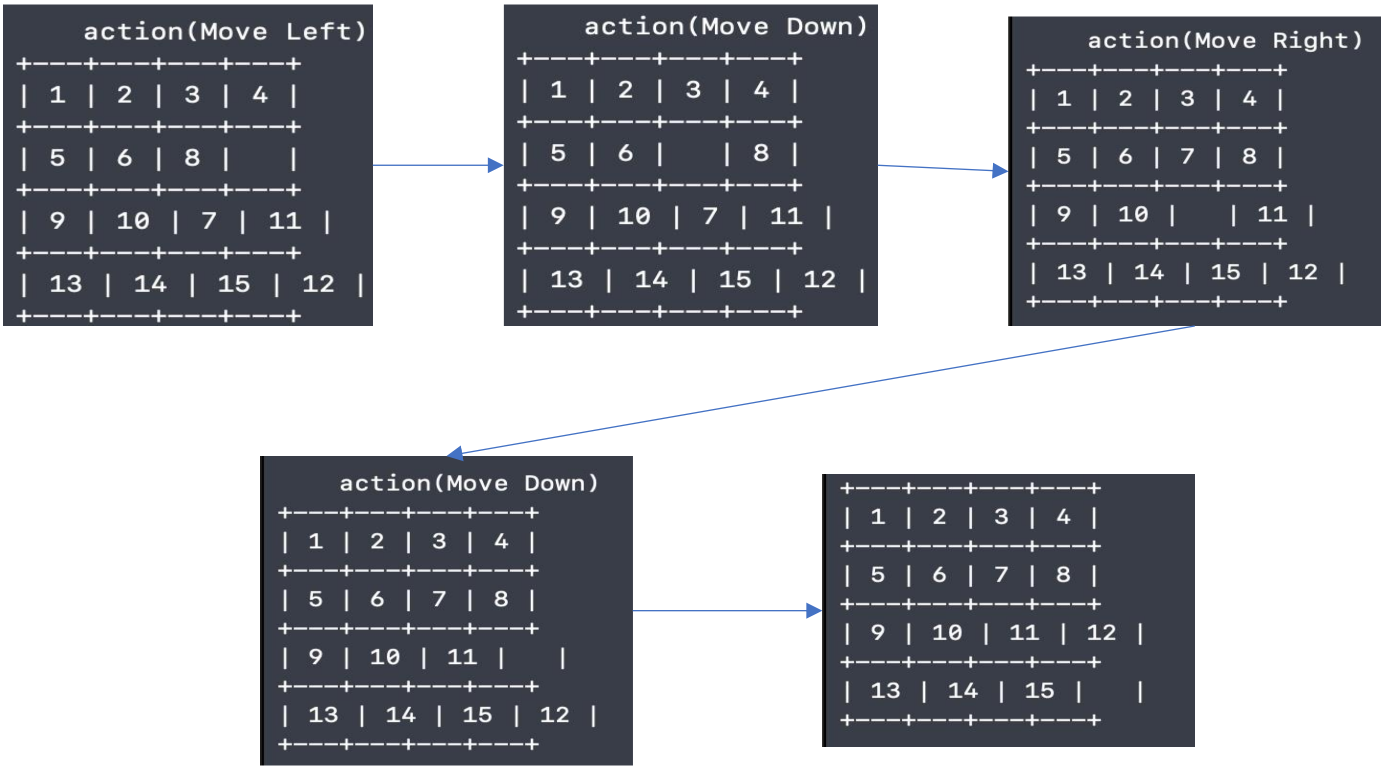
It took 1.09 seconds to find the solution

For 4x4 Board:

Assume that user entered an initial state

{{1, 2, 3, 4},{5, 6, 8, 0},{9, 10, 7, 11}},{13, 14, 15, 12}} which requires 4 steps.

1) **Breadth-First Search:**




```
The number of searched nodes is : 49

The number of generated nodes is : 68

The number of generated nodes in memory is : 68

THE COST PATH IS 4.00.
```

It took 3.31 seconds to find the solution

2)Uniform-Cost Search:

```
 action(Move Left)
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 5 | 6 | 8 |   |
+---+---+---+---+
| 9 | 10 | 7 | 11 |
+---+---+---+---+
| 13 | 14 | 15 | 12 |
+---+---+---+---+
```

```
 action(Move Down)
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 5 | 6 |   | 8 |
+---+---+---+---+
| 9 | 10 | 7 | 11 |
+---+---+---+---+
| 13 | 14 | 15 | 12 |
+---+---+---+---+
```

```
 action(Move Right)
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 5 | 6 | 7 | 8 |
+---+---+---+---+
| 9 | 10 |   | 11 |
+---+---+---+---+
| 13 | 14 | 15 | 12 |
+---+---+---+---+
```

```
 action(Move Down)
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 5 | 6 | 7 | 8 |
+---+---+---+---+
| 9 | 10 | 11 |   |
+---+---+---+---+
| 13 | 14 | 15 | 12 |
+---+---+---+---+
```

```
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 5 | 6 | 7 | 8 |
+---+---+---+---+
| 9 | 10 | 11 | 12 |
+---+---+---+---+
| 13 | 14 | 15 |   |
+---+---+---+---+
```

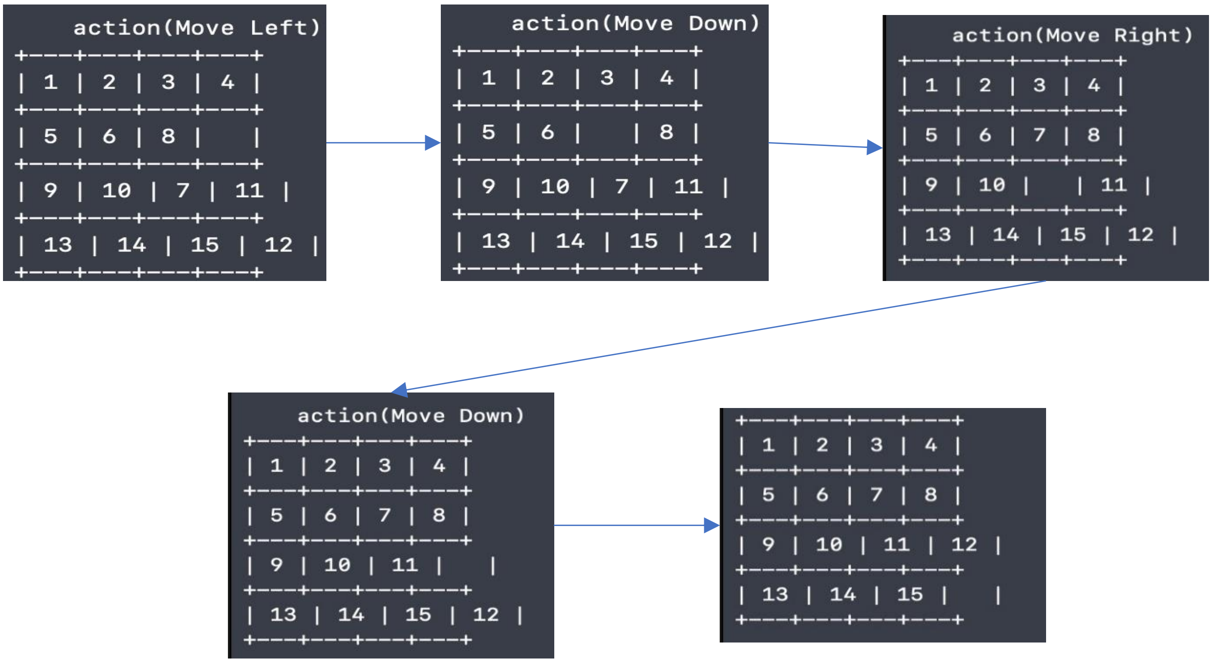
```
The number of searched nodes is : 49

The number of generated nodes is : 155

The number of generated nodes in memory is : 155

THE COST PATH IS 4.00.
```

It took 7.34 seconds to find the solution

3)Depth-First Search : Depth-first search blindly follows one branch to its maximum depth (and often continues exploring even after finding a solution), revisiting states without tracking them and thus suffering exponential node expansions. As a result, its total expanded-node count and solution path cost become unpredictable and can far exceed the optimal length. For this instance, where the optimal path cost is only 4 moves, naive DFS without cycle checks failed to find the solution.

4) Depth Limited Search

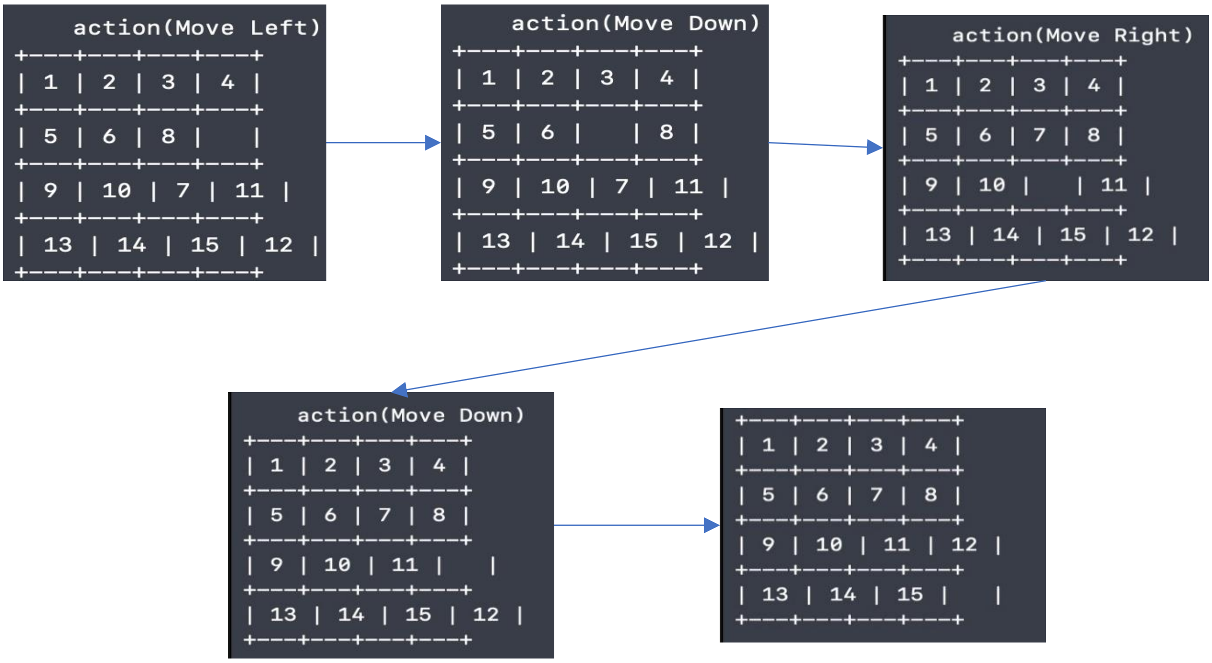
In this algorithm, selecting a depth of 3 or less will fail to find a solution because no solution exists at that depth. For the given example, a depth of 4 has been chosen.




```
The number of searched nodes is : 22

The number of generated nodes is : 28

The number of generated nodes in memory is : 11

THE COST PATH IS 4.00.
```

It took 2.48 seconds to find the solution

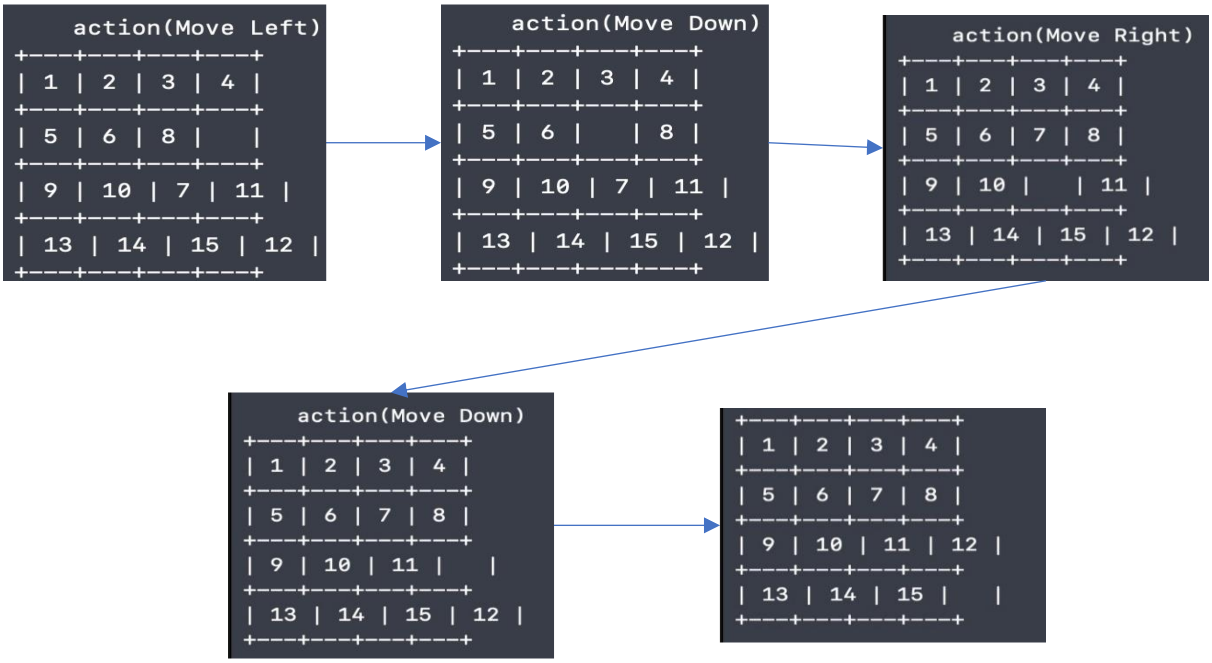
5)Iterative Deepening Search




```
The number of searched nodes is : 61

The number of generated nodes is : 75

The number of generated nodes in memory is : 11
The goal is found in level 4.

THE COST PATH IS 4.00.
```

It took 6.24 seconds to find the solution

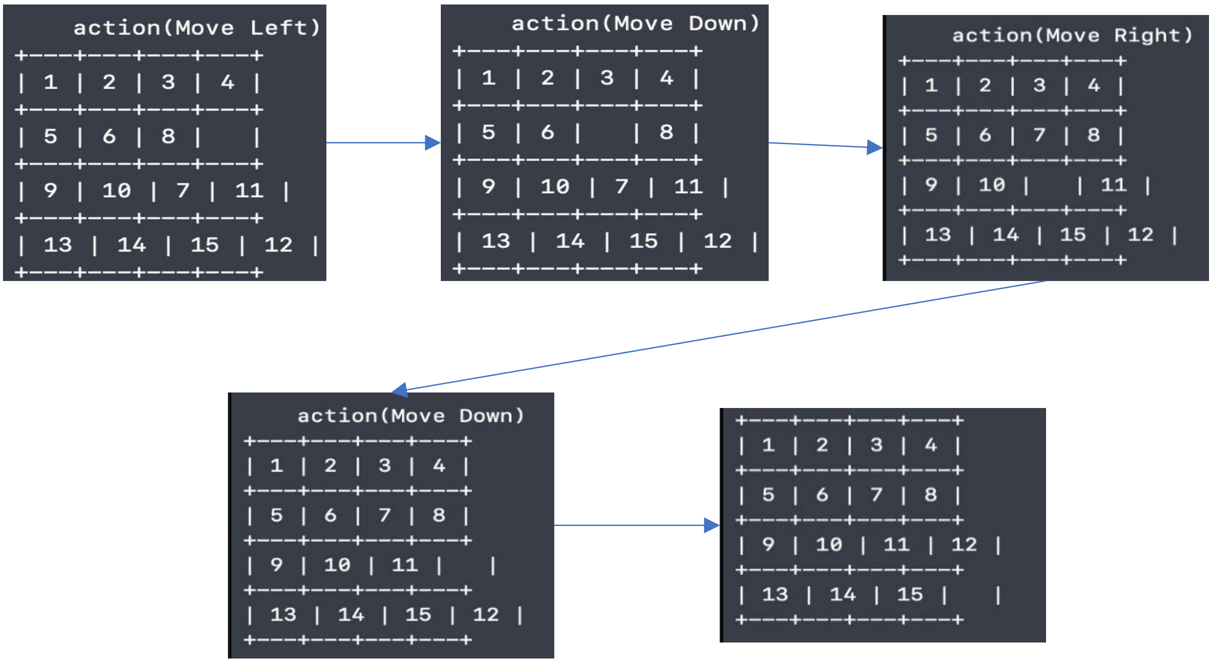
6)Greedy Search




```
The number of searched nodes is : 12

The number of generated nodes is : 14

The number of generated nodes in memory is : 14

THE COST PATH IS 4.00.
```

It took 1.79 seconds to find the solution

7)A* Search

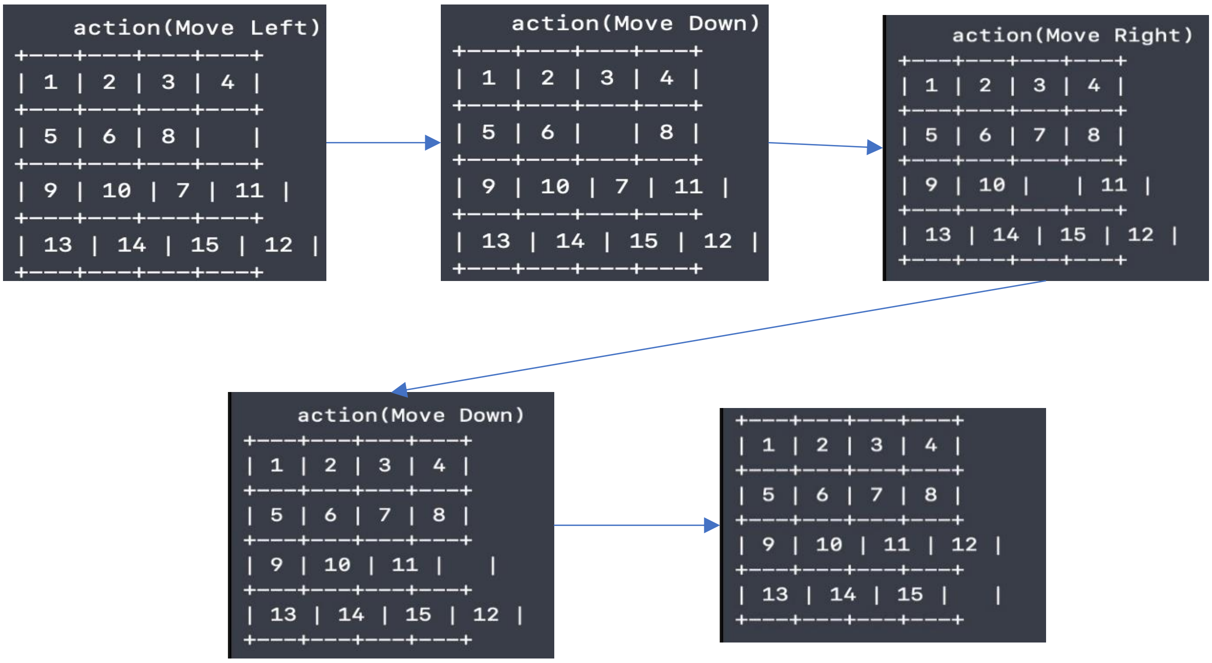



```
The number of searched nodes is : 5

The number of generated nodes is : 15

The number of generated nodes in memory is : 15

THE COST PATH IS 4.00.
```

It took 1.01 seconds to find the solution

8) Generalized A* Search

In generalized A*, the **critical α value** is defined as the **smallest weight coefficient** that ensures the goal node's $f_\alpha(n) = g(n) + \alpha\, h(n)$ remains **lower than that of every other node** in the frontier—thereby guaranteeing that the goal node is expanded first. In this case, the critical α was found to be **0.61**.




```
The number of searched nodes is : 5

The number of generated nodes is : 15

The number of generated nodes in memory is : 15

THE COST PATH IS 4.00.
```

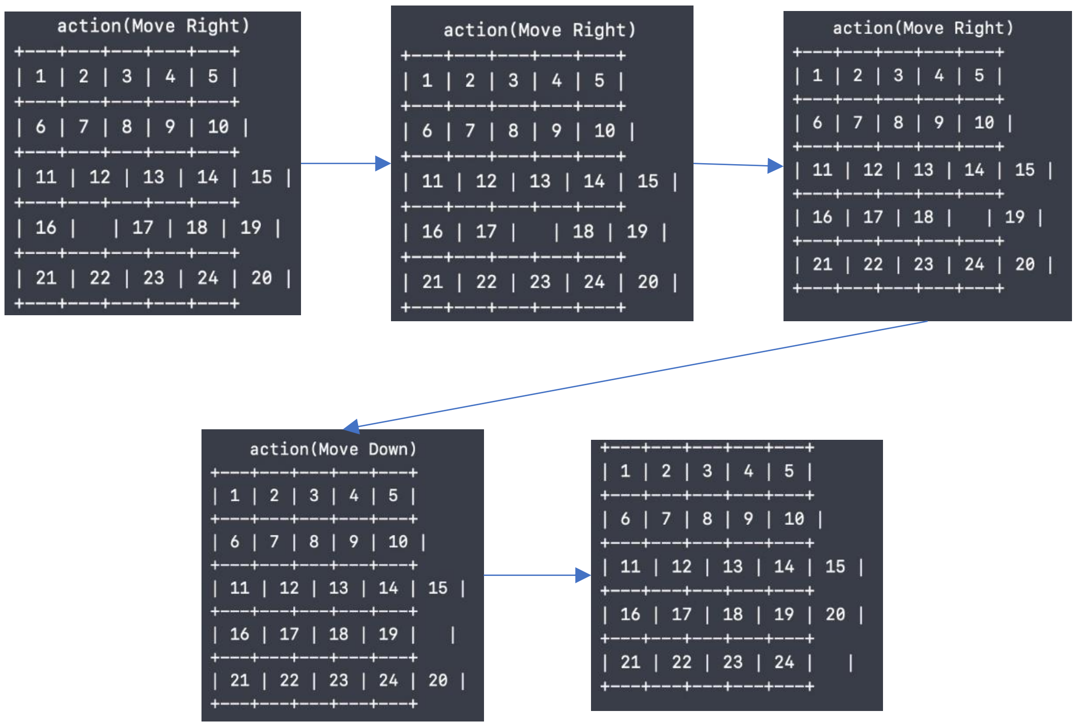
It took 1.01 seconds to find the solution

For 5x5 Board:

Assume that user entered an initial state

{{1, 2, 3, 4, 5},{6, 7, 8, 9, 10},{11, 12, 13, 14, 15}},{16, 0, 17, 18, 19},{21, 22, 23, 24, 20}} which requires 4 steps.

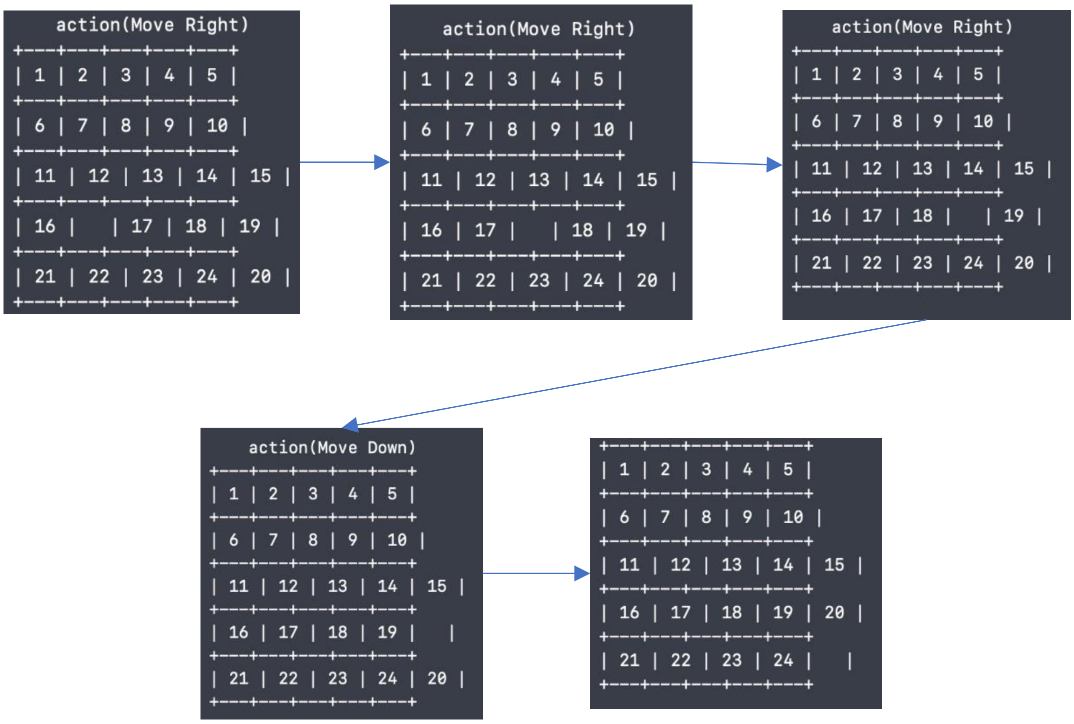
1) Breadth-First Search:




```
The number of searched nodes is : 91

The number of generated nodes is : 126

The number of generated nodes in memory is : 126

THE COST PATH IS 4.00.
```

It took 6.3 seconds to find the solution

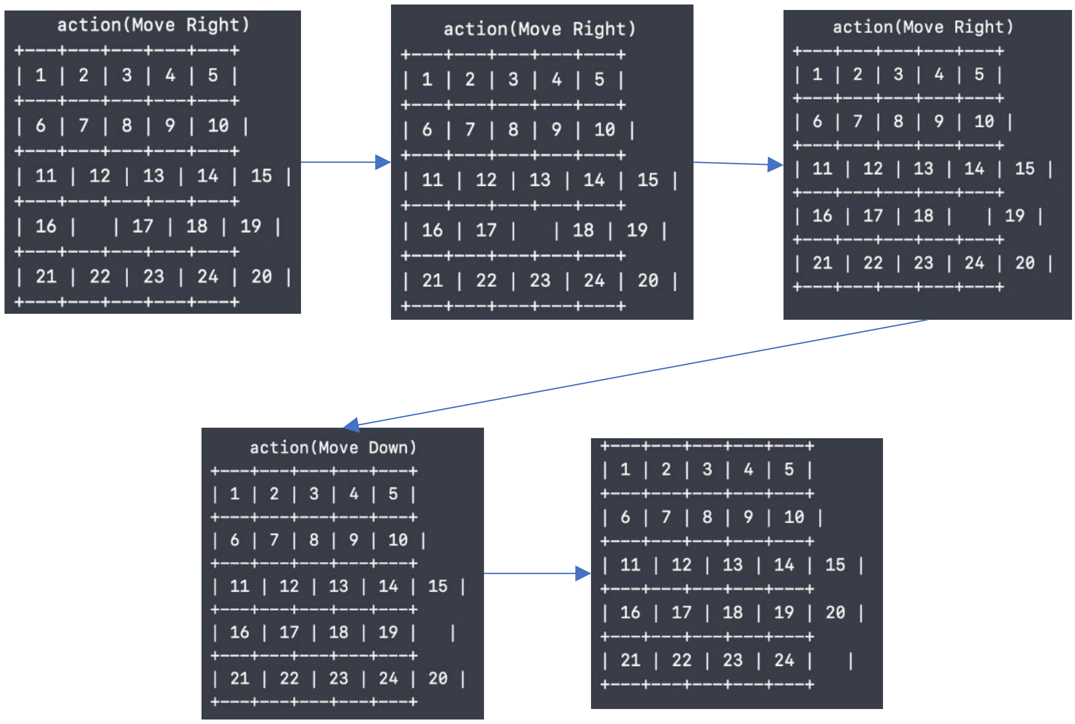
2) Uniform-Cost Search:




```
The number of searched nodes is : 91

The number of generated nodes is : 307

The number of generated nodes in memory is : 307

THE COST PATH IS 4.00.
```

It took 21.50 seconds to find the solution

3)Depth First Search

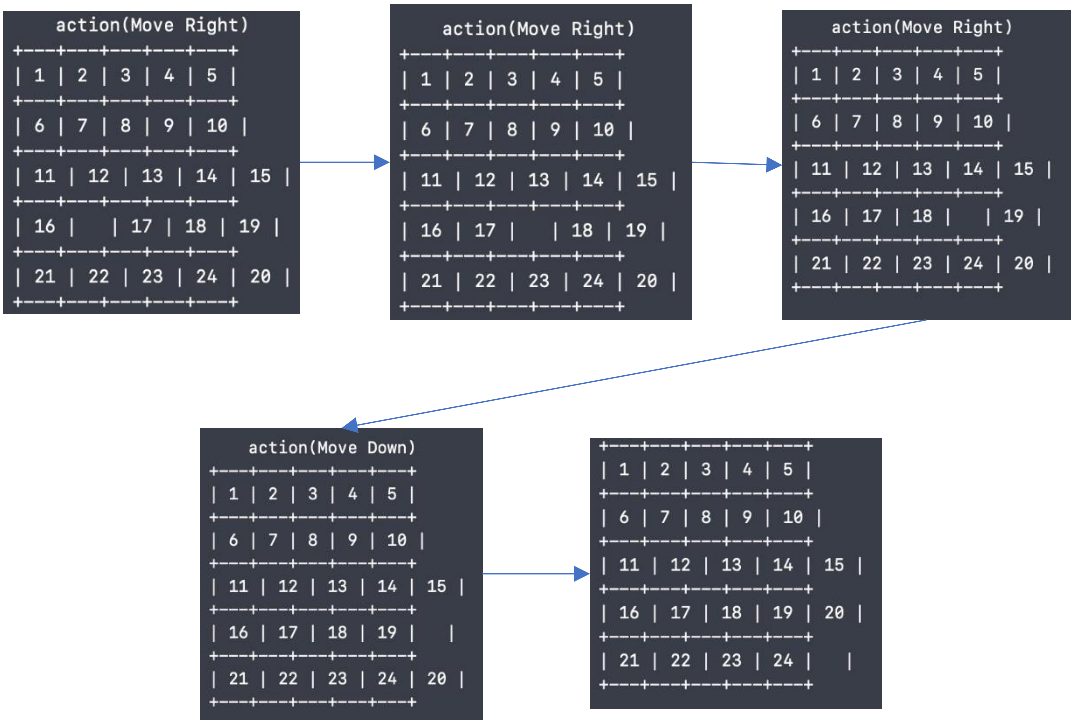



```
The number of searched nodes is : 13

The number of generated nodes is : 15

The number of generated nodes in memory is : 13

THE COST PATH IS 4.00.
```

It took 2.28 seconds to find the solution

4)Depth Limited Search

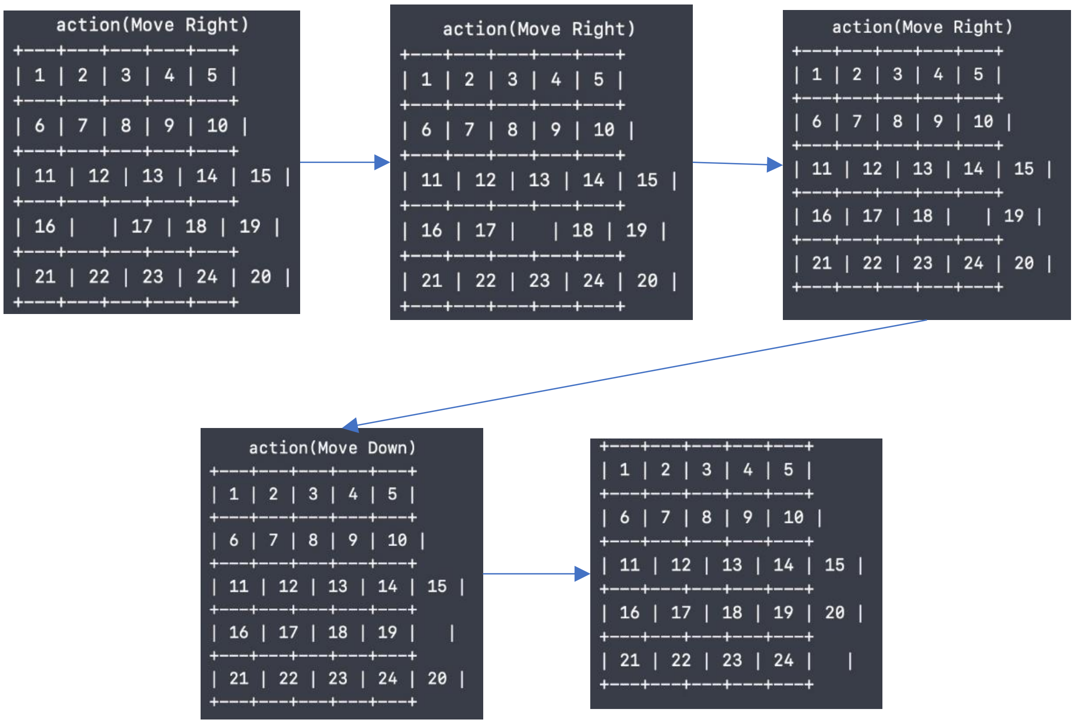
In this algorithm, selecting a depth of 3 or less will fail to find a solution because no solution exists at that depth. For the given example, a depth of 4 has been chosen.




```
The number of searched nodes is : 13

The number of generated nodes is : 15

The number of generated nodes in memory is : 13

THE COST PATH IS 4.00.
```

It took 1.26 seconds to find the solution

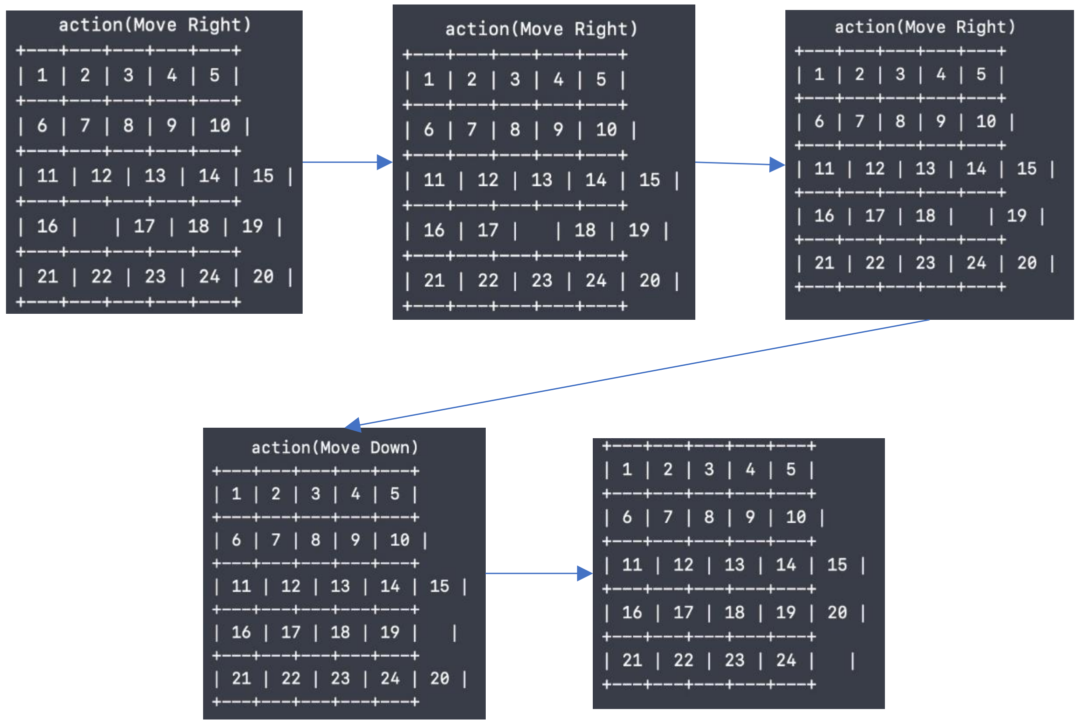
5)Iterative Deepening Search




```
The number of searched nodes is : 71

The number of generated nodes is : 87

The number of generated nodes in memory is : 13
The goal is found in level 4.

THE COST PATH IS 4.00.
```

It took 1.38 seconds to find the solution

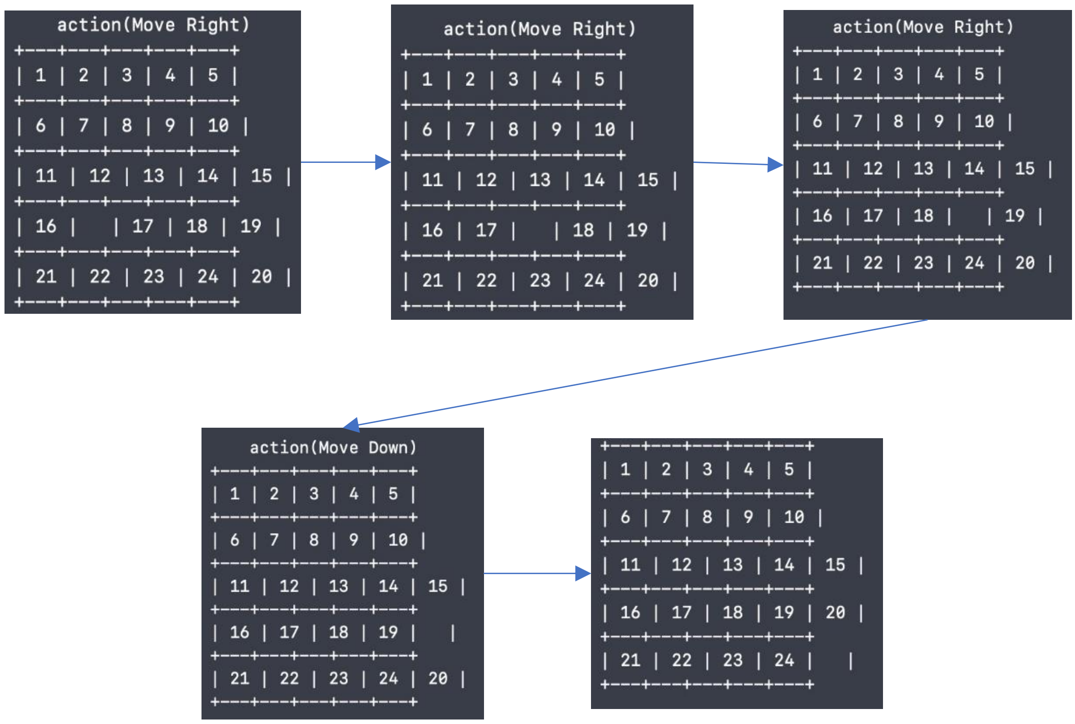
6)Greedy Search




```
The number of searched nodes is : 13

The number of generated nodes is : 15

The number of generated nodes in memory is : 13

THE COST PATH IS 4.00.
```

It took 1.28 seconds to find the solution

7)A* Search




```
The number of searched nodes is : 5

The number of generated nodes is : 16

The number of generated nodes in memory is : 16

THE COST PATH IS 4.00.
```

It took 1.13 seconds to find the solution

8) Generalized A* Search

In generalized A*, the **critical α value** is defined as the **smallest weight coefficient** that ensures the goal node's $f_\alpha(n) = g(n) + \alpha\, h(n)$ remains **lower than that of every other node** in the frontier—thereby guaranteeing that the goal node is expanded first. In this case, the critical α was found to be **0.61**.

```
action(Move Right)
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
| 6 | 7 | 8 | 9 | 10 |
+---+---+---+---+---+
| 11 | 12 | 13 | 14 | 15 |
+---+---+---+---+---+
| 16 |    | 17 | 18 | 19 |
+---+---+---+---+---+
| 21 | 22 | 23 | 24 | 20 |
+---+---+---+---+---+
```

```
action(Move Right)
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
| 6 | 7 | 8 | 9 | 10 |
+---+---+---+---+---+
| 11 | 12 | 13 | 14 | 15 |
+---+---+---+---+---+
| 16 | 17 |    | 18 | 19 |
+---+---+---+---+---+
| 21 | 22 | 23 | 24 | 20 |
+---+---+---+---+---+
```

```
action(Move Right)
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
| 6 | 7 | 8 | 9 | 10 |
+---+---+---+---+---+
| 11 | 12 | 13 | 14 | 15 |
+---+---+---+---+---+
| 16 | 17 | 18 |    | 19 |
+---+---+---+---+---+
| 21 | 22 | 23 | 24 | 20 |
+---+---+---+---+---+
```

```
action(Move Down)
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
| 6 | 7 | 8 | 9 | 10 |
+---+---+---+---+---+
| 11 | 12 | 13 | 14 | 15 |
+---+---+---+---+---+
| 16 | 17 | 18 | 19 |    |
+---+---+---+---+---+
| 21 | 22 | 23 | 24 | 20 |
+---+---+---+---+---+
```

```
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
| 6 | 7 | 8 | 9 | 10 |
+---+---+---+---+---+
| 11 | 12 | 13 | 14 | 15 |
+---+---+---+---+---+
| 16 | 17 | 18 | 19 | 20 |
+---+---+---+---+---+
| 21 | 22 | 23 | 24 |    |
+---+---+---+---+---+
```

```
The number of searched nodes is : 5

The number of generated nodes is : 16

The number of generated nodes in memory is : 16

THE COST PATH IS 4.00.
```

It took 1.13 seconds to find the solution

Conclusion

- We implemented eight search strategies—from uninformed (BFS, UCS, DFS/DLS/IDS, Greedy) to informed (A*, Generalized A*)—for solving the 8-Puzzle.
- Manhattan Distance + Linear Conflict proved to be the most efficient heuristic, dramatically reducing both node expansions and runtime.
- A* (α=1) and Generalized A* delivered optimal solutions faster than uninformed methods, while Greedy was fastest but not always optimal.
- Depth-First variants (DFS, DLS, IDS) trade memory for speed but require careful depth limits or cycle checks.
- Important note: On larger puzzles (e.g. a 4×4 board), plain Depth-First Search can easily fall into infinite loops unless you enforce a depth bound or track visited states.
- The critical α in Generalized A* ($\approx$0.61) offers a tunable balance between exploration cost and heuristic bias.

Future Work

- Experiment with more advanced heuristics (e.g. pattern databases).
- Scale to larger puzzles (15-Puzzle, 24-Puzzle) with parallel or GPU-accelerated search.
- Integrate bidirectional or symmetry-breaking optimizations for further speed-ups.